FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Diplomarbeitspräsentation

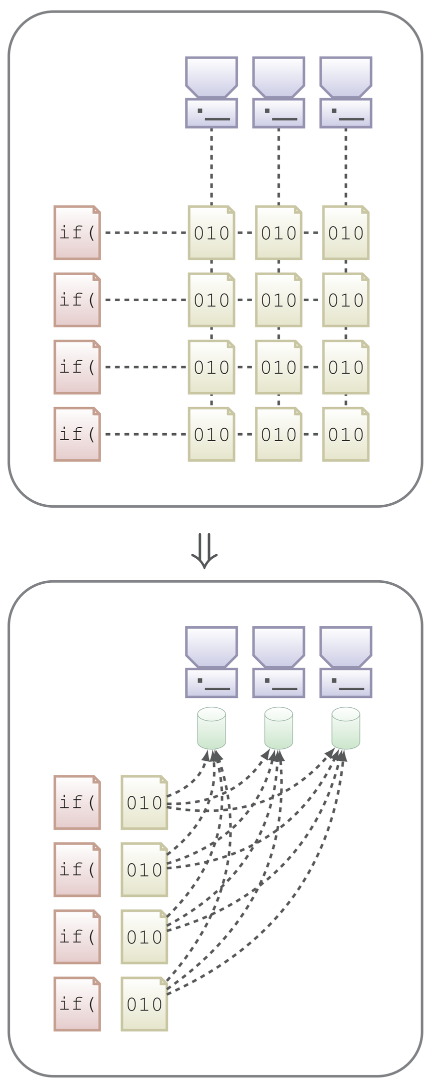
Optimization Framework for the CACAO VM

Masterstudium:
Computational Intelligence

Josef Eisl

Technische Universität Wien
Institut für Computersprachen
Arbeitsbereich: Programmiersprachen und Übersetzer
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Virtual Machines



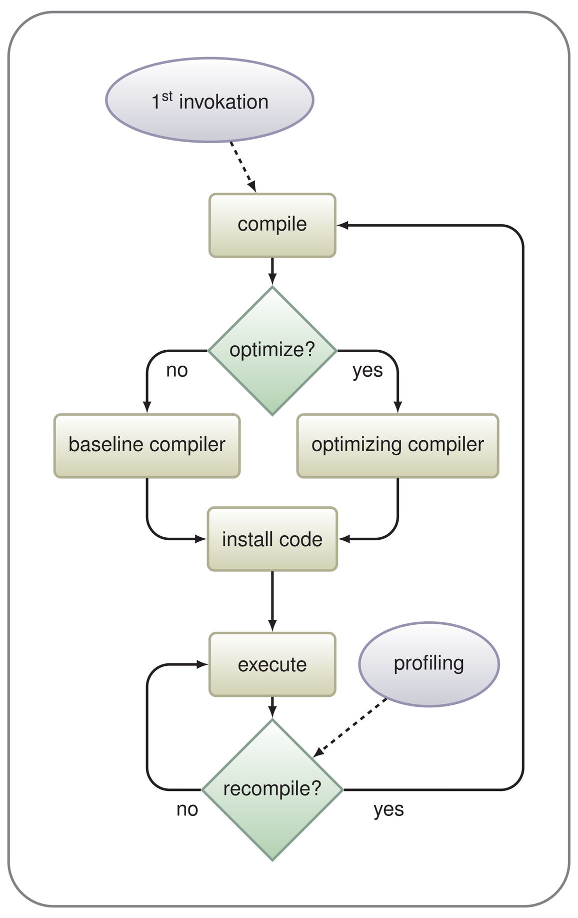
- **Why Virtual Machines?**
  - **Portability:** Reduce the number of program images from $P \times M$ to $P + M$.
  - **Runtime System:** Virtual machines provide a runtime system to the application, for instance garbage collection or dynamic loading.
  - **Security:** Program images can be signed and verified prior execution.
- **Program representation:**
  - **Bytecode:** A dedicated ahead-of-time compiler converts the program source into a portable, low-level representation (Java Bytecode, Microsoft CLI, Pascal p-code).
  - **Source Code:** The source code of the program is supplied to the virtual machine. Usually the VM compiles the source into bytecode. Commonly used for “scripting languages” (JavaScript, Python, Ruby).
- **Execution:**
  - **Interpretation:** The virtual machine *simulates* the instructions of the program.
  - **Compilation:** The program is *just-int-time* (JIT) *compiled* into native machine code.

Adaptive Optimization

- **Why Adaptive Optimization?**
  - **Interprete Everything:** Program execution is too slow.
  - **Compile Everything:** Program startup is too slow.
- **Solution:**
  - **Profile** the runtime behavior and find frequently executed parts.
  - **Compile** these *hot methods* with a high optimization level to generate better performing machine code.
  - **Redirect** all calls to the optimized machine code.

The CACAO VM

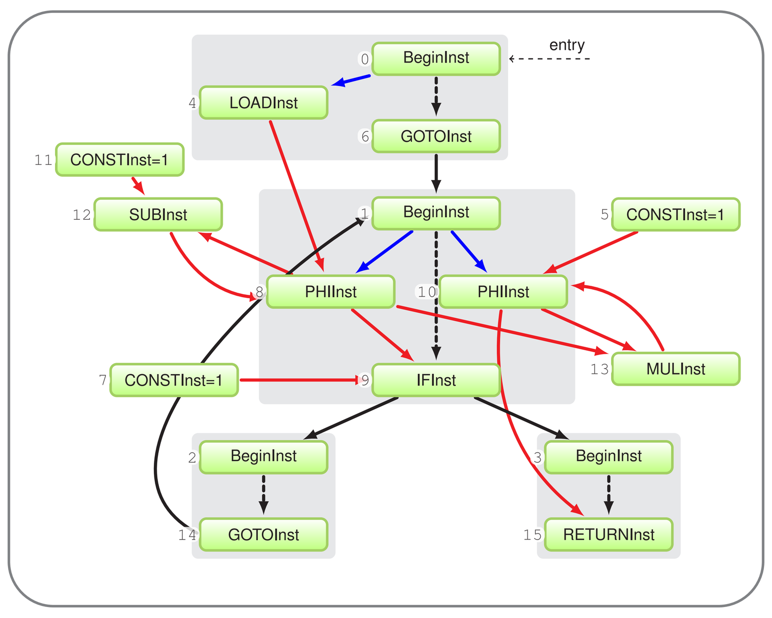
- **Virtual Machine for Java bytecode** [1]
- **JIT-only approach:**
  - No interpreter but a fast *baseline* compiler.
  - Better performance than interpretation but inferior code quality compared to heavy weight optimizing compilers.
- **Problem:** Baseline compiler not suitable for elaborate optimizations.
  - Tuned for low compilation latency (integrated passes, simple data structures).
- **Solution:** Dedicated Optimization Framework.
  - Independent optimizing compiler.
  - Focused on program optimization and analysis.
  - Make compiler-passes easy to implement.



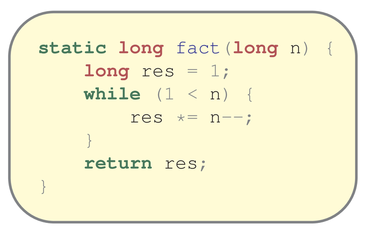
Pass Manager

- **Modular pass infrastructure** with a clean interface.
  - *Pass objects* encapsulate data and computation.
  - Data exchange via *Pass objects* – no global data structures.
- **Automatic pass scheduling** based on interdependencies.
  - Optional passes can be inserted on demand.
  - Dynamic optimization profiles based on runtime information.

High-level IR



- **Graph-based IR** [2]
- **SSA-form:**
  - No *variables* or *register*.
  - No destructive assignment.
  - Instructions represent values.
- **Less restrictive:**
  - No explicit basic blocks.
  - No fixed schedule:
    - Floating nodes.
- **Edge types:**
  - *data edge*
  - *control-flow edge*
  - *scheduling edge*
- **Goal:**
  - Support pass development.
  - Targeted on common tasks:
    - Data-flow graph (DFG) traversal.
    - Control-flow graph (CFG) traversal.
- **Additional scheduling required:**
  - Arrange floating instruction in basic blocks (Global Scheduling).



Low-level IR

- **Classic intermediate representation:**
  - Explicit *basic blocks* (already in linear order).
  - Basic blocks consist of a list of *instructions*.
    - Ideally one LIR instruction models one machine instruction.
  - Explicit operands for data transfer:
    - Virtual and physical registers, virtual and physical stack-slots, constants.
- **Relaxed SSA-properties:**
  - *Dominance* property and *single reaching* definition (φ-nodes).
  - Possible to express SSA-violating constraints (two-address instruction, fixed register operand) and still use simple algorithms.
- **Focused on register allocation and code generation.**

Passes

- **High-level passes:**
  - SSA construction, loop analysis, dominator analysis, basic block/global/instruction scheduling, lowering to LIR
- **Low-level passes:**
  - Lifetime analysis, linear scan register allocation, code generation

Results

☹ *Compile time*: 30-50 times slower than the baseline compiler.
- Tuning potential. Not yet optimized towards this goal.

☹ Higher *memory consumption*.

☺ *Comparable codesize* even without advanced optimizations.

☺ Optimizations *are* easy to implement, for instance:
- **Deadcode elimination:** single iteration over all instructions.
- **Constant folding:** single recursive traversal over instructions.
- *No additional analysis required!*

References

- Andreas Krall.
  Efficient JavaVM Just-in-Time Compilation. 1998.
- Cliff Click and Michael Paleczny.
  A Simple Graph-Based Intermediate Representation. 1995.

Kontakt: zapster@complang.tuwien.ac.at